Methodologies for Knowledge-Based Software Engineering

Michael Lowry
Recom Technologies, Inc.
Artificial Intelligence Research Branch
Mail Stop 269-2
NASA Ames Research Center
Moffett Feild, CA 94035-1000

NASA Ames Research Center
Artificial Intelligence Research Branch

Technical Report FIA-93-07

March, 1993

To Appear in ISMIS '93

Methodologies for Knowledge-Based Software Engineering

Michael R. Lowry

AI Research Branch, MS 269-2
NASA Ames Research Center / Recom Technologies
Moffett Field, CA 94035

Abstract. As the science of knowledge representation and automated reasoning advances, AI has the potential to radically change the artifacts, methodologies, and life cycles of software engineering. The most significant change will be when problems are formalized at the level of specifications rather than programs. This will greatly facilitate software reuse and modification. Achieving this potential requires overcoming many technical challenges, particularly the semi-automated synthesis of efficient and correct programs from specifications. The first part of this paper describes several methodologies for program synthesis and compares their ability to control the combinatorial explosion inherent in automated reasoning.

As knowledge-based software engineering matures and increasingly automates the software engineering life cycle, software engineering resources will shift toward knowledge acquisition and the automated reuse of expert knowledge for developing software artifacts. The second part of this paper describes methodologies for expanding the software life cycle to the knowledge life cycle.

1 Introduction

In the early sixties large software projects, such as those undertaken for NASA's Apollo program, forced software engineering to mature from an ad hoc endeavor practiced by small teams of programmers to a structured engineering discipline. Structured programming methodologies were developed to cope with the complexities of managing, specifying, designing, and implementing large software systems. Structured designs were captured through hand drawn diagrams depicting everything from project decomposition to data and control flow. CASE (computer-aided software engineering) emerged in the eighties when it became economically feasible to computerize structured programming by providing graphical user interfaces to manipulate these diagrams.

KBSE (knowledge-based software engineering) is a much more ambitious endeavor than current approaches to CASE. The key observation is that the current practice of modern software engineering lacks the sound mathematical basis characterizing other engineering disciplines. This limits the complexity of software systems that can be constructed with a high degree of reliability. Formal methods, the application of mathematical logic to software engineering, is just beginning to have an impact on real software engineering practice. The goal of KBSE is nothing less than the computerization of formal methods for all phases of the software life cycle [8].

KBSE addresses the essential tension between problem specification and efficient solution implementation. This tension makes it difficult to modify and reuse programs,

since efficient code incorporates constraints from all parts of a problem specification in the optimization of individual program fragments. Hence, local incremental changes to a problem specification often require extensive non-local changes to optimized code. Modification of production quality code is so time consuming that maintenance costs currently dominate software life cycle resources. Furthermore, reuse of production quality code has been difficult to achieve. Advances in programming language and compiler technology have raised the level of programming abstractions, but have not addressed the essential difference between optimized code suitable for efficient computation and formal problem specifications suitable for reuse and modification. The current paradigms for programming languages cannot in principle bridge this gap, because to guarantee compiler performance the peephole on the source code used in optimizing machine language code must be limited.

KBSE bridges this gap by introducing a new software development paradigm: problems are first formalized at the level of the declarative semantics of an application domain, and then semi-automatically transformed to the operational semantics of an efficiently compileable programming language [8]. Formal methods provide the mathematical basis for this transformation. Automated reasoning provides the means for carrying out the transformation. By raising the level at which problems are formalized, modification and reuse will be greatly facilitated. Furthermore, by introducing formal artifacts earlier in the software life cycle, mechanized support can be provided for the full spectrum of software engineering activities, from requirements engineering to validation and maintenance.

Evolutionary improvements of current software engineering methodologies can be achieved with existing KBSE technology. Particularly promising are domain-specific program synthesis tools and re-engineering tools to modify and maintain existing code. However, achieving the full KBSE paradigm will require many technical advances. Foremost are search control for automated reasoning, and interactive assistance in requirements formalization and validation.

Raising the level at which problems are formalized from the programming level to the specification level will eliminate many conceptual and design errors. These errors can cost over a hundred times more to fix during the testing phase than simple coding errors [1]. This is one major motivation for applying formal methods even without computer-aided assistance. However, even at the specification level, formalization is a difficult process, and many of the most costly software errors can be traced back to the transformation from informal requirements to specifications. New methods for specification validation are needed. AI programming environments have already contributed to one method, rapid prototyping, in which executable specifications are developed in a very high level programming language and then validated interactively with end users. Other AI approaches to specification validation are described in [17, 18, 24] and their references.

This paper first describes several methodologies for program synthesis with an emphasis on search control, drawing upon the literature and the author's own work. It then contrasts knowledge-based methodologies for software engineering with methodologies based on CASE, and describes the evolution from the software engineering life cycle to the knowledge engineering life cycle.

2 Methodologies for Program Synthesis

2.1 Constructive Theorem Proving

Soon after Robinson [27] developed resolution as the first practical means for automated theorem proving in predicate logic, it was applied to automatic program synthesis. Green [6] and Waldinger [33] demonstrated the generation of small programs such as sorting algorithms through constructive proofs from specifications of the form:

$\forall x \exists y \, \text{Precondition}(x) \rightarrow \text{Postcondition}(x, y)$

In this specification schema x is a vector of input variables, y is a vector of output variables, and Precondition(x) is a formula constraining the input variables. A constructive proof binds y to a term which makes the specification a theorem. If this term is composed of functions in the programming language and the only variables in the term are input variables, then the term represents a functional program. The inference process is similar to logic programming: first the universally quantified variables in x are replaced by unique constants, the formula Precondition(x) is asserted over these constants, and then Postcondition(x,y) is negated and resolution is repeatedly applied until a refutation is derived. The program term is built up through unification with the output variables y. Recursive and iterative constructs are derived through inductive proofs using inductive schemata or through additional inference rules. Manna and Waldinger [19] later developed an elegant nonclausal variation suitable for manually controlled derivations.

Initially it was hoped that advances in generic theorem proving strategies would sufficiently control search to enable automated derivations to scale up to large problems. While impressive progress was made during the early seventies [3], generic resolution strategies were never able to mitigate combinatorial explosion effectively. Program synthesis often requires deep reasoning; generating recursive and iterative programming constructs through inductive proofs considerably expands the combinatorial explosion. In retrospect, it is unlikely that general purpose theorem-proving strategies will ever be sufficient to control the combinatorial search inherent in automated program synthesis.

2.2 Program Transformations

An alternative approach to program synthesis is incremental transformation of specifications to implementations through program transformations, i.e. oriented rewrite rules. In its purest form, the transformational approach is formalized by the semantics of conditional equational logic. In its more restricted variants the transformational approach can greatly reduce search [23]. It is also more adaptable than theorem proving to less formal knowledge engineering approaches, and can be viewed as an extension of current compiler technology. Transformations are typically oriented from higher level specification constructs to lower level implementation constructs, thus providing an overall direction to the search. Sets of rewrite rules are characterized by properties such as termination and confluence (guaranteed termination in a unique normal form). The Knuth-Bendix completion procedure [12], given an appropriate weighting scheme for terms and a set of rewrite rules, will add rewrite rules until the set becomes confluent. The Knuth-Bendix completion procedure is not guaranteed to terminate, and generally works only on small sets of rules and for restricted kinds of weighting schemes.

Despite the well-behaved search properties for restricted variants of program transformations, obtaining the same problem solving power as constructive theorem proving ultimately requires addressing the same combinatorial search issues. One way of increasing problem solving capability is to make the conditions for applying an inference rule more complex; but as the complexity is increased the amount of inference required typically increases exponentially. Furthermore, introducing general capabilities for producing recursive and iterative constructs requires expanded capabilities such as folding [4], in which a rewrite rule is reversed to introduce the application of a function from an instance of its definition. This reversal of rewrite rule orientation leads to a combinatorial explosion of possibilities. In general, as the scope of a transformation system is expanded to encompass a larger set of possible programs as output, the search space expands drastically.

2.3 Manually Guided Program Synthesis/Verification

At the opposite extreme to totally automated search control, several early systems (e.g.[2]) had the user select each primitive step of the inference or transformation process. Initially it was hoped that this approach would be a viable means for mechanically assisted program synthesis or verification. However, the sheer number of steps required made this approach infeasible outside of research settings or in applications such as avionics requiring extreme reliability. It was far easier to develop a program by hand than guide an inference system through the large number of primitive steps.

The problems with totally automated search control using generic strategies and the other extreme of manual guidance of primitive inference/transformation rules has led to methodologies centered on human/computer partnerships and reuse. These include the intelligent assistant approach in which humans make strategic decisions while the computer carries out bounded searches, reuse of generic programs or derivations, and encoding of tactical and strategic program design knowledge. Each of these methodologies introduces interacting knowledge representation and automated reasoning issues.

2.4 Intelligent Programming Assistant

Floyd [5] presented an early vision of an intelligent programming assistant, in which the computer kept track of clerical details while the human made the important strategic decisions. A key issue is developing representations for program derivations that are human comprehensible and machine manipulable. The decision making also has to be factored to limit the search carried out by automated reasoning while presenting meaningful strategic decisions for the human user. These constraints rule out certain technologies such as clausal resolution.

The programmer's assistant project at MIT, spanning the years 1973 to 1992, was an influential effort particularly in the area of re-engineering. The main achievements in the early years were the development of the plan formalism, a language independent representation for programs and programming knowledge, and demonstration of KBEmacs, an editor for manipulating programs in this formalism.

The plan formalism represented programs as flowcharts with explicit data and control flow arcs. The main innovation of the plan formalism was support for programming clichés, which are reusable algorithmic fragments (such as enumeration over a file) that were engineered to correspond to expert human programming knowledge. Analyzers were developed for several programming languages that recognized instances of clichés

in program text. The combination of analyzers and translators between the plan formalism and program text enabled significant re-engineering capabilities, such as modification of programs at the level of programming clichés and improved translation of programs between programming languages via abstraction to the plan formalism and then reimplementation in the target programming language [34].

However, the plan formalism lacked the semantic basis to provide generality and power; reasoning was carried out by ad hoc procedures. This limited the feasibility of extending the capabilities of KBEemacs. To address these limitations Rich [25] formalized the plan formalism into the plan calculus and then developed CAKE [26], a layered automated reasoning system. CAKE is a careful integration of different automated reasoning capabilities (e.g. truth maintenance, propositional reasoning, equality, and types) that appears as an active knowledge base for software artifacts. CAKE's automatically invoked inference procedures are constrained to run in polynomial time; user queries can invoke more time consuming reasoning procedures. Significant new capabilities for the programmer's apprentice were implemented on top of the plan calculus and CAKE, including a debugging assistant [14] and a requirements assistant (RA) [28]. The RA is a good example of the interactive problem formalization assistance that can be provided through KBSE: the RA notified a user when it detected ambiguity, contradiction, incompleteness, or inaccuracy in an evolving requirements specification.

Several lessons can be learned from the evolution of MIT's programmer assistant project. First, although a knowledge engineering approach is useful in the initial development of a representation meaningful to humans, achieving generality and power requires a semantically well defined formal representation with semantically well founded inference procedures. Without these, the implicit assumptions which facilitate an ad hoc approach become limitations hindering the expansion of a KBSE system. These factors will limit the expansion of current CASE systems, because of their shallow representations and ad hoc reasoning procedures. Second, developing the automated reasoning capabilities to support a formal representation requires significant engineering to avoid combinatorial explosion.

2.5 Replaying Program Derivations

An alternative to reusing high-level generic program fragments such as clichés is the reuse of program derivations. This approach spans the range from rote replay of derivations to derivational analogy [20]. Program derivation reuse is particularly appealing because it has the potential to support the incremental modification of specifications by rederiving efficient implementations through replay of the original derivation. When the replay system encounters part of the derivation which is no longer applicable, then it transfers control to the user.

Derivational analogy replay systems have been successfully applied in domains such as VLSI design where the mapping from the input of the derivation system to the output is localized; that is, each part of the output is attributable to localized parts of the input. However, as discussed earlier, optimized code must potentially incorporate constraints from all parts of a specification. For this reason, substantial parts of the original program derivation might no longer be applicable after an incremental change in specification. To date, most derivational analogy replay systems for program synthesis have operated on representations of the enablement structure of transformation or inference rules. There is typically no representation in the derivation record of the purpose for applying a transformation in meeting a performance goal for the optimized code (however, see [35]).

Thus there is insufficient information for making good analogies to other derivations when parts of the original derivation are no longer applicable. For these reasons, derivational analogy replay systems have had little more success so far than rote replay systems in program synthesis.

2.6 Design Analysis

While generic theorem proving or transformational strategies have had limited success in automating program synthesis, a more promising methodology is to develop efficient tactics for controlling automated reasoning for particular classes of software artifacts. Each tactic can be viewed in itself as a special purpose program synthesizer. However, because tactics are control programs for general purpose inference mechanisms, they can be easily combined. The following describes one methodology for developing tactics within the context of parameterized theories and algebraic specifications. The example used is the development of a tactic for synthesizing local search algorithms, more details can be found in [15].

Design analysis is a methodology for formalizing both the structural properties common to a class of software artifacts and the genetic properties common to their derivations [30]. This formalization is then used to develop a design tactic that automatically designs an artifact in this class given a specification of its behavior. Design analysis formalizes intrinsic structural properties rather than properties specific to a particular programming language or application domain. By abstracting away these particular concerns, the resulting formalization is more broadly applicable. The objective is to find a general mathematical characterization of the structure of a class while at the same time capturing the features that provide search guidance for designing artifacts in a class.

The first step of design analysis for algorithm synthesis is to study many examples of a naturally defined class of algorithms. For example, local search algorithms, also referred to as hill–climbing algorithms, are a natural class in which a feasible solution to an optimization problem is iteratively improved by searching a neighborhood of the solution for a better solution, and stopping when no neighboring solution is better. The second step of design analysis is to extract the features and structural constraints characterizing that class of algorithms. The neighborhood structure determines the properties of a local search algorithm : exact neighborhood structures guarantee that local optimums are global optimums, while the weaker condition of reachability guarantees that all feasible solutions for a given input are mutually reachable. Reachability is a necessary condition for variants of local search that can bactrack out of local optimums, such as simulated annealing, to converge on global optimums. The third step is to formalize this characterization in a theory. The theory of neighborhood structures for local search algorithms is an extension of the theory for optimization problems.

A basic problem is specified by defining a set of inputs **D**, a set of outputs **R**, an operation **I** that maps legal inputs to true, and an operation **O** that maps input/output pairs to true when the output is a feasible solution to the input. A basic problem specification is a tuple **B= <D,R,I,O>**.

An optimization problem is specified by extending a basic problem specification with an ordering relation in which all pairs of feasible solutions are comparable. All such ordering relations can be formulated as a cost function that maps feasible solutions to a totally ordered set. For most problems the cost function maps feasible solutions to the integers, rationals, or reals. The totally ordered set is denoted $\langle \Re, \leq \rangle$, where $\Re$ is the set

and $\leq$ is the total order relation. Thus an optimization problem is specified through a tuple **Opt=⟨D,R,I,O,ℜ,≤,cost⟩**.

A local search theory **LS = ⟨Opt,N⟩** is specified by an optimization problem and a neighborhood relation. Three axioms, two being optional, constrain the neighborhood relation, which is a ternary relation between an input and two elements of the output domain. First, each feasible solution is in its own neighborhood, so that for any legal input the neighborhood relation is a reflexive relation on feasible outputs (Axiom LS1). If the neighborhood structure is exact, then the local search theory will be called exact (Axiom LS2). Likewise, if the neighborhood structure is reachable, the local search theory will be called reachable (Axiom LS3). A local search theory for a particular optimization problem is defined by a mapping from the components of abstract local search theory to definitions of objects, functions and relations in the problem domain. More formally, the mapping is a theory interpretation, which means that the abstract axioms are true when they are mapped to the problem domain theory. Abstract local search theory is defined as follows:

Sorts **D,R,ℜ**

Operations

$\mathbf{I}: \mathbf{D} \rightarrow$ boolean

$\mathbf{O}: \mathbf{D} \times \mathbf{R} \rightarrow$ boolean

$\mathbf{cost}: \mathbf{D} \times \mathbf{R} \rightarrow \Re$

$\leq: \Re \times \Re \rightarrow$ boolean

$\mathbf{N}: \mathbf{D} \times \mathbf{R} \times \mathbf{R} \rightarrow$ boolean

$\text{Optimal}(x,y) \equiv \forall(y')\mathbf{O}(x,y') \Rightarrow \mathbf{cost}(x,y) \leq \mathbf{cost}(x,y')$

Axioms

LS1: **Reflexive Neighborhood**

$\forall(x,y)\ \mathbf{I}(x) \wedge \mathbf{O}(x,y) \Rightarrow \mathbf{N}(x,y,y)$

LS2: **Exact Neighborhood**

$\forall(x,y)\ \mathbf{I}(x) \wedge \mathbf{O}(x,y) \wedge \left[\forall(y')\ \mathbf{O}(x,y') \wedge \mathbf{N}(x,y,y') \Rightarrow \mathbf{cost}(x,y) \leq \mathbf{cost}(x,y')\right] \Rightarrow \text{Optimal}(x,y)$

LS3: **Reachable Neighborhood**

$\forall(x,y,y')\ \mathbf{I}(x) \wedge \mathbf{O}(x,y) \wedge \mathbf{O}(x,y') \Rightarrow \mathbf{N}^*(x,y,y')$

where $\mathbf{N}^*$ is the reflexive and transitive closure of **N**:

$\forall(x,y)\mathbf{N}^0(x,y,y) \equiv \mathbf{I}(x) \wedge \mathbf{O}(x,y)$

$\forall(k \in Nat; x,y,y')\mathbf{N}^{k+1}(x,y,y') \equiv \exists(z)\mathbf{O}(x,z) \wedge \mathbf{N}^k(x,y,z) \wedge \mathbf{N}(x,z,y')$

$\forall(x,y,y')\mathbf{N}^*(x,y,y') \equiv \exists(k \in Nat)\mathbf{N}^k(x,y,y')$

To derive a local search algorithm for a particular optimization problem, a partial mapping from this abstract local search theory to the components of an optimization problem is first created. Constraints for a suitable neighborhood relation are then derived by instantiating the abstract neighborhood axioms with these components. The main part of the design tactic is to derive the definition of a neighborhood relation from these constraints in terms of the problem domain. Once the neighborhood relation is defined,

an initial algorithm can be derived by instantiating a program schema with the components of the derived local search theory. This high-level algorithm can be further refined with optimization tactics such as partial deduction [13] and finite differencing [21].

Formalizing the structure of a class of software artifacts is by itself usually insufficient for providing mechanized design assistance; it is also necessary to formalize the structure of the derivations. In the example of local search algorithms, the axioms for reachability and exactness defined above are too general to avoid combinatorial explosion in automated reasoning. The axioms for reachability require induction over the transitive closure of neighborhoods, which can be difficult for automated theorem provers. The exact neighborhood axiom, as stated, does not provide sufficient structure for determing its satisfaction for most problems. (Typically, proofs for exact neighborhoods are done through reduction to problems with known exact neighborhoods such as linear programming, or through lemmas about convex functions.)

To provide heuristic adequacy for guiding derivations, various specializations of the general structure are derived. For local search algorithms whose neighborhoods are reachable but not necessarily exact, most neighborhoods for efficient local search algorithms can be described as natural perturbations of data structures: [The key step in deriving a local search algorithm is the] "... selection of a neighborhood or a class of neighborhoods, and this is tied to the notion of a 'natural' perturbation of a feasible solution" ([22] pg. 469). The theory of groups and group actions provides the mathematical basis for formalizing natural perturbations.

A natural perturbation neighborhood is defined for a data structure by a set of permutations and a group action mapping these permutations to perturbations of each instance of the data structure. Thus the neighborhoods for all instances of the data structure are similar extentionally and have the same intentional description based on the set of permutations. A permutation is any one-to-one (and hence invertible) function from some set of objects to the same set. The closure of this set of permutations under composition together with the group action defines three interrelated structures: a group of permutations, the mutually reachable data structures, and the invariant properties of mutually reachable data structures.

The specialization of reachable neighborhoods to natural perturbations entails only two restrictions on the reachable neighborhoods axiomatized in the abstract theory of local search. First, neighborhoods are required to be symmetric, that is if y is in x's neighborhood then x is in y's neighborhood. Most local search algorithms satisfy this condition. This condition ensures that if z is reachable from w then w is reachable from z. Second, the neighborhoods of all feasible solutions are similar; they have the same intensional description in terms of the set of permutations. These two restrictions are sufficient to enable the tools of group theory to be used in developing reachable neighborhood structures for a wide variety of optimization problems. The mathematics and proofs are fully developed in [15].

Specializing reachable neighborhoods to natural perturbation neighborhoods considerably simplifies automated reasoning. In particular, it is no longer necessary to do an inductive proof on the transitive closure of neighborhoods: reachability is ensured if and only if the invariant properties of a natural perturbation neighborhood are equivalent to the feasibility constraints for problem solutions. If the invariants are stronger than the feasibility constraints then some feasible solutions would not be reachable from other feasible solutions. If the invariants are weaker than the feasibility constraints then some feasible solutions would be mapped to infeasible solutions.

The local search design tactic developed in this approach matches a problem specification to a library theory whose invariants are equivalent or weaker than the feasibility constraints, and then specializes the library theory if the invariants are weaker. The library theories are defined for general set theoretic data structures, such as ordered sequences, as explained below. Reasoning about invariant properties and feasibility constraints provides a computationally tractable method of matching and then specializing theories in a library to problem specifications. The theorem prover does not have to reason directly about the second order reachability axioms - this has already been done by the creator of the library theories.

Library theories are based on basic neighborhoods which are the subclass of natural perturbation neighborhoods in which the permutations are restricted to be all the transpositions of some underlying set, that is, permutations in which only two elements are interchanged. For example, one basic neighborhood structure for an ordered sequence is defined by all the transpositions of the indices of the sequence. Basic neighborhoods are typically overly general for any particular problem; the design tactic first matches a problem specification to a basic neighborhood and then specializes the basic neighborhood. The current library of parameterized natural perturbation neighborhood theories consists of a half dozen basic neighborhood definitions, which include specifications of their invariants. A basic neighborhood has the following definition schema as a ternary relation, where y,y' are neighboring feasible solutions with respect to input x, and i,j are the elements that are transposed:

$\lambda\, x, y, y'.\exists(i, j \in S)\ y' = \mathbf{Action}(x, y, i, j)$

A local search library theory for a basic neighborhood consists of the basic neighborhood definition and definitions for the other components of a local search theory. It is presented as a mapping of the following form from abstract local search theory to a set of definitions:

LS - basic theory
$\mathbf{D} \mapsto \mathit{datatype1}(\alpha)$
$\mathbf{R} \mapsto \mathit{datatype2}(\alpha)$
$\mathbf{I} \mapsto \lambda x.\mathbf{P}(x)$
$\mathbf{O} \mapsto \lambda x, y.\ \mathbf{Invariant}(x, y)$
$\mathbf{N} \mapsto \lambda x, y, y'.\ \exists(i, j \in \mathbf{F}(x))\ y' = \mathbf{Action}(x, y, i, j)$

For example, the following mapping from abstract local search theory defines the basic neighborhood structure for same-sized subsets of a given finite set S. The size of the subsets are a constant size m, the elements which are transposed are the elements of the finite set:

LS - subset theory
$\mathbf{D} \mapsto \mathit{set}(\alpha) \times \mathit{integer}$
$\mathbf{R} \mapsto \mathit{set}(\alpha)$
$\mathbf{I} \mapsto \lambda S, m.\ m \leq \mathrm{size}(S)$
$\mathbf{O} \mapsto \lambda S, m, y.\ y \subseteq S \wedge \mathrm{size}(y) = m$
$\mathbf{N} \mapsto \lambda S, m, y, y'.\ \exists(i, j) i \in (S - y) \wedge\ j \in y \wedge\ y' = (y \cup \{i\}) - \{j\}$

This theory can be matched to a wide range of problems including the class of ACS (additive cost subset) problems defined by Savage. A typical example is the minimal

spanning tree problem (MST), which is to find a minimally weighted subset of edges in a graph that span the nodes of the graph without any cycles.

Given the specification of an optimization problem, the local search design tactic first matches the problem specification to a library theory for a basic neighborhood, and then specializes the library theory by finding necessary conditions on transpositions to ensure that feasible solutions are transformed to better feasible solutions. The design tactic takes the following steps; each step is a well defined inference problem with a manageable search space; the overall effect is to replace a large search space with a sequence of smaller search spaces:

1. Retrieve and match basic neighborhood theories from the library indexed by the type of feasible solution. A theory matches a problem specification if the invariants of the theory are necessary conditions of the feasibility constraints of the specification.
2. Determine necessary preconditions on the transpositions that ensure that a feasible solution is perturbed to a feasible solution.
3. Determine necessary preconditions on the transpositions that ensure that a feasible solution is perturbed to a better feasible solution.
4. (Optional step for deriving exact local search algorithms) Determine necessary conditions for a local optimum to be a global optimum.
5. Instantiate a program schema with the components of the theory where the derived preconditions on transpositions are guards on the application of a transposition.

Proofs for the correctness of the tactic and a detailed description illustrated with the derivation of the simplex algorithm can be found in [15]. This design tactic was developed as an extension of the KIDS system [31]. A generalization of the methods for matching library theories to problem specifications is presented in [32].

2.7 Summary of Methodologies for Program Synthesis

The previous subsections have reviewed various methodologies for automated or semi-automated program synthesis. In order to generate production quality code from high level problem specifications, a program synthesis system should be able to incorporate constraints from all parts of a specification into each program fragment. This cannot be achieved by translation systems which restrict the window on the source code considered in optimizing the output code.

The general dilemma is that the number of possible programs that can be generated from a given specification is combinatorially explosive. General purpose methods have not been found for efficiently searching this space for efficient programs; it is unlikely any such universal method exists. However, as described above, currently there are some effective techniques for capturing the knowledge used by expert human programmers and applying it in automated fashion. Even setting aside the issue of program correctness, it is worthwhile capturing this knowledge within a formal representation with semantically well-founded inference methods. This avoids the limitations that arise when trying to generalize and expand ad hoc methods. To avoid intractable searches during automated program synthesis, it is necessary for knowledge representations and inference methods to be carefully designed in tandem to limit search spaces. It is likely that as continued progress is made in automated program synthesis, improvements in formal knowledge representations will be driven as much by considerations of making inference tractable as by theoretical considerations in mathematical logic.

3 From the Software Life Cycle to the Knowledge Life Cycle

The previous section described various methodologies for program synthesis. While automated program synthesis is necessary to raise software engineering from the programming level to the specification level, it is only one component of the KBSE paradigm. As KBSE matures and increasingly automates the software engineering life cycle, software engineering resources will increasingly shift toward knowledge acquisition and the automated reuse of knowledge for developing software artifacts. This section describes how the various components of KBSE could interact.

Knowledge, like software, has its own characteristic life cycle. The knowledge life cycle is the maturation of design knowledge for an application domain from the initial research stage to the cookbook engineering stage. Knowledge-based design tools can provide support at stages of the knowledge life cycle that are not well supported with conventional software design tools. Furthermore, knowledge-based design tools have the potential of significantly compressing the knowledge life cycle.

One principle objective of KBSE is to compress the software life cycle with knowledge-based tools. By its very nature, knowledge-based design depends on the maturity of knowledge about an application domain. As knowledge about an application domain is developed by scientists and engineers, the design process for that domain makes a transition from creative and innovative design to routine and cookbook design. Different kinds of design tools are appropriate at different stages of the knowledge life cycle. One of the main anticipated advantages of KBSE is the ability to leverage the expertise of scientists and engineers by capturing their design knowledge at all stages of the knowledge life cycle.

An example of the knowledge life cycle is the development of the theory and technology for designing parsers used in compilers and other language-processing systems. Early parsers in the late fifties were ad hoc systems. Not only was there a lack of a theory of parsing to guide their design, there was not even a theory of grammars that specified the function of a parser. Thus the design of early parsers was creative: both the structure and function were unknown and ill-defined. The development of the BNF formalism in the early sixties clarified the function of a parser: to produce a trace of the BNF rules used to generate a text string from the text string itself. At this stage the design of parsers became innovative: the function was known, but the structure of possible solutions was still unexplored.

The mid sixties to the mid seventies witnessed rapid development of the theory and technology for parsing. First, recursive descent parsing was formalized, enabling parser development to become a routine design task. While routine, the design of these early recursive descent parsers required the configuration of a set of procedures, i.e., configurational design. By the late sixties, several table-driven parsers were developed: operator precedence, LL parsing, and LR parsing. (These early parsing formalisms did not handle left recursion, which required the development of LALR parsing.) Designing a table-driven parser is now a routine parametric design process, that is, the output of a design is a set of parameters for a specialized representation. When knowledge about an application domain becomes this advanced, then automated design tools can be readily developed with conventional software technology. Hence in the late sixties and seventies, parser generators were developed that take a specification of a grammar and automatically generate the parameters for a table-driven parser.

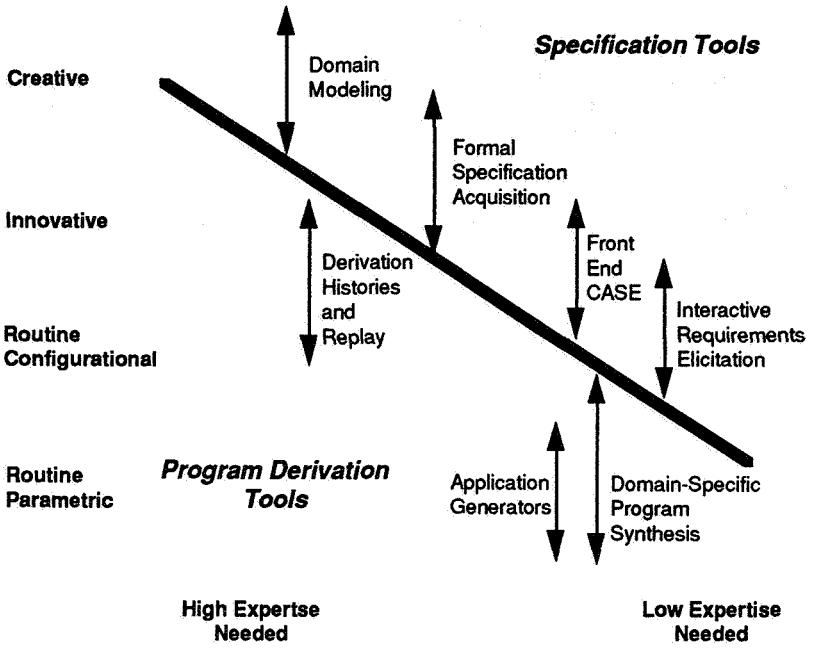

Figure 1. Spectrum of knowledge-based tools in the knowledge life cycle.

Figure 1 shows how knowledge-based design tools fit into the spectrum from creative design to routine parametric design. The horizontal axis denotes the level of human expertise required to use the tool, while the diagonal line separates specification tools from program derivation tools. The figure illustrates that knowledge-based tools can be used much earlier in the knowledge life cycle than current CASE tools. Domain modeling tools use a formal modeling language to express knowledge about an application domain. This knowledge can be used for different operational goals throughout the software life cycle, from requirements engineering to re-engineering. Domain modeling is the first step in moving from creative design to routine design. Because domain modeling is essentially the formalization of domain knowledge it requires a high degree of expertise, both in the application domain and in knowledge representation formalisms. During the innovative phase of the knowledge life cycle, general purpose interactive program synthesis systems could be used to explore the solution space for an application domain. Many of these general purpose systems have facilities for recording, editing, and replaying derivation histories. These replay facilities might enable users with less expertise than the original designer to develop derivations for similar specifications [20]. Given domain knowledge from a domain modeling tool, formal specifications can be developed and incrementally modified with tools such as ARIES [9].

Figure 1 also illustrates that knowledge-based tools can be employed by users with lower levels of expertise than required for current CASE tools. Front end CASE tools enable designers to define and edit software abstractions like data flow diagrams during

the initial stages of system design. However, because these CASE tools lack application domain knowledge they require more expertise and provide lower levels of verification and simulation capabilities than can be provided with knowledge-based tools. A good example of a knowledge-based specification tool is the WATSON system [11], which interactively elicits and validates requirements for new telephone features (such as call waiting) from telephony engineers using domain level scenarios. Back end CASE tools such as application generators (e.g., parser generators) are currently widely used for the final stages of coding, particularly in commercial data processing. They consist of a menu driven or application language front end and a template-driven code generator back end. As such they are suitable for routine parametric design. In contrast, domain-specific program synthesis systems also can be used for routine configurational design and have a high level user interface that reduces the expertise needed by an end-user. Because systems like ELF [29] and SYNAPSE [10] combine template driven code generation with more powerful AI semantic processing techniques and transformation rules, they can tackle routine configurational design in addition to parametric design. They also produce more optimal code than an application generator.

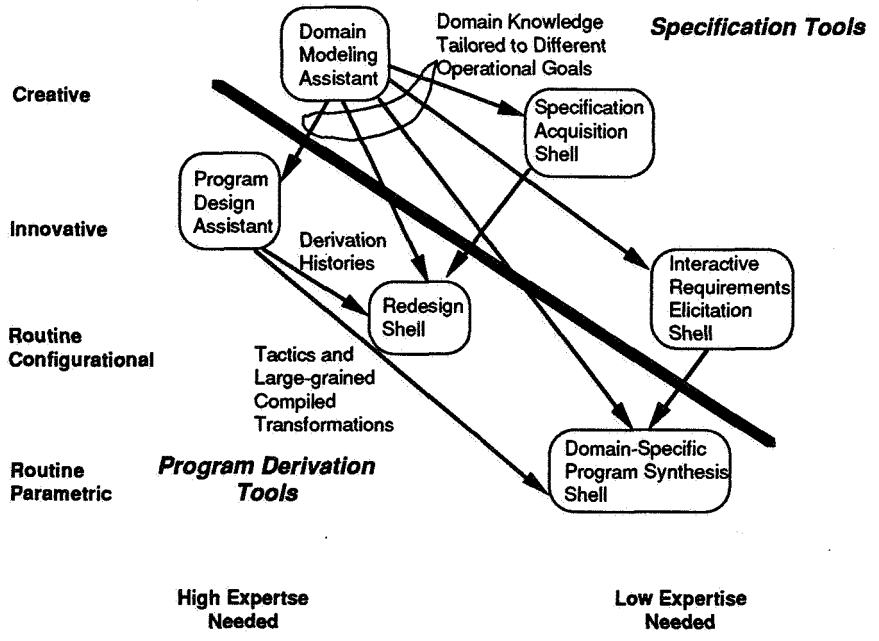

Figure 2. Transfer and reuse of expertise in the KBSE paradigm.

Knowledge-based technology, by providing an active medium for communication of knowledge, can also potentially compress the knowledge life cycle. In the absence of major advances in machine discovery, the development of design knowledge will continue to be a human-intensive process requiring high levels of expertise. However, knowledge-based tools could assist human research scientists and engineers in the development of this knowledge. These tools could then compile and transfer this

knowledge into shells that do interactive requirements elicitation, redesign, specification acquisition, and domain-specific program synthesis.

Figure 2 illustrates this transfer of expertise using knowledge-based subsystems; a more detailed exposition can be found in [16]. The domain modeling assistant and program design assistant would be used by scientists and engineers during the creative and innovative stages of the knowledge life cycle. The specification acquisition shell would support future system analysts in developing formal system specifications, while the redesign shell would enable system developers to construct software systems rapidly by editing and replaying derivation histories developed with a program design assistant.

For end-users with low expertise, requirements elicitation shells would use domain knowledge to develop interactively formal specifications of their requirements using informal examples. Domain-specific program synthesis shells would then synthesize a system meeting these requirements. Note that while a system developer with intermediate expertise could be expected to understand and manipulate derivation histories to develop a software system, an end user with low expertise would require totally automatic program synthesis. Thus a domain specific synthesis shell would require more highly compiled control knowledge for controlling software derivation than a redesign shell, i.e., tactics or large-grained compiled transformations as opposed to interpreting and manipulating derivation histories.

4 Conclusion

Knowledge based software engineering is based on research spanning over two decades. Significant commercial applications are likely within this next decade, particularly as industrial pilot projects in domain specific program synthesis and re-engineering mature. Interest in formal methods will also spur development of the field. However, to achieve the full paradigm requires many technical advances in knowledge representation and automated reasoning. This paper has described various methodologies for making automated reasoning tractable for program synthesis. Further improvements are likely to require that knowledge representations for program design expertise be developed in tandem with automated reasoning methods.

The paradigm of the knowledge life cycle can help to clarify the role of knowledge-based software engineering tools and guide their development. Different kinds of knowledge-based tools are appropriate at different stages of the knowledge life cycle. Furthermore, knowledge-based tools can expedite the transfer of expertise from research scientists and engineers and thus compress the knowledge life cycle.

References

1. B.W. Boehm: Software Engineering Economics. Englewood: Prentice Hall1981
2. M. Broy, P. Pepper: Program Development as a Formal Activity. IEEE Trans. on Software Eng. 7(1), 14-22 (1981)
3. C.L. Chang, R.C.T. Lee: Symbolic Logic and Mechanical Theorem Proving. New York: Academic Press 1973
4. J. Darlington: An Experimental Program Transformation and Synthesis System. Artificial Intelligence 16, 1-46 (1981)

5. R. Floyd: Toward Interactive Design of Correct Programs. In C. Rich, R.C. Waters (eds.): Artificial Intelligence and Software Engineering. Los Altos: Morgan Kaufmann 1986, pp. 331-334

6. C. Green: Application of Theorem Proving to Problem Solving. IJCAI (1969)

8. C. Green, D. Luckham, R. Balzer, T. Cheatham, C. Rich: Report on a Knowledge-Based Software Assistant. In C. Rich, R.C. Waters (eds.): Artificial Intelligence and Software Engineering. Los Altos: Morgan Kaufmann 1986, pp. 337-428

9. W.L. Johnson, M.S. Feather: Using Evolution Transformations to Construct Specifications. In M.R. Lowry, R.D. McCartney (eds.): Automating Software Design. Cambride, MA: AAAI/MIT Press 1991, pp. 65-92

10. E. Kant, F. Daube, W. MacGregor, J. Wald: Scientific Programming by Automated Synthesis. In M.R. Lowry, R.D. McCartney (eds.): Automating Software Design. Cambride, MA: AAAI/MIT Press 1991, pp. 169-206

11. V.E. Kelly, U. Nonnenmann: Reducing the Complexity of Formal Specification Acquisition. In M.R. Lowry, R.D. McCartney (eds.): Automating Software Design. Cambride, MA: AAAI/MIT Press 1991, pp. 41-64

12. D.E. Knuth, P.B. Bendix: Simple Word Problems in Universal Algebras. In J. Leach (ed.): Computational Problems in Abstract Algebra. Pergammon Press 1970, pp. 263-298

13. J. Komorowski: Synthesis of Programs in the Partial Deduction Framework. In M.R. Lowry, R.D. McCartney (eds.): Automating Software Design. Cambride, MA: AAAI/MIT Press 1991, pp. 377-404

14. R.I. Kuper: Dependency-directed localization of software bugs. Tech. Report 1053 MIT AI Lab, 1989

15. M.R. Lowry: Automating the Design of Local Search Algorithms. In M.R. Lowry, R.D. McCartney (eds.): Automating Software Design. Cambride, MA: AAAI/ MIT Press 1991, pp. 515-546

16. M.R. Lowry: Software Engineering in the Twenty-First Century. In M.R. Lowry, R.D. McCartney (eds.): Automating Software Design. Cambride, MA: AAAI/ MIT Press 1991, pp. 627-654

17. M.R. Lowry, R. Duran: Knowledge-based Software Engineering. In A. Barr, P.R. Cohen, E.A. Feigenbaum (eds.): The Handbook of Artificial Intelligence, Vol. 4. Reading, MA: Addison-Weseley, 1989

18. M.R. Lowry, R.D. McCartney (eds.): Automating Software Design. Cambride, MA: AAAI/MIT Press 1991

19. Z. Manna, R. Waldinger: A Deductive Approach to Program Synthesis. ACM Trans. on Prog. Lang. and Sys. 2(1): 90-121 (1980)

20. J. Mostow: Design by Derivational Analogy: Issues in the Automated Replay of Design Plans. Artificial Intelligence 40, 119-184 (1989)

21. R. Paige, S. Koenig: Finite Differencing of Computable Expressions. ACM Transactions on Programming Languages and Systems 4(3): 402-454 (1982)

22. C.H. Papidimitriou, K. Steiglitz: Combinatorial Optimization. Englewood, N.J.: Prentice-Hall 1982

23. U.S. Reddy: Design Principles for an Interactive Program-Derivation System. In M.R. Lowry, R.D. McCartney (eds.): Automating Software Design. Cambride, MA: AAAI/MIT Press 1991, pp. 453-482

24. C. Rich, R.C. Waters (eds.): Artificial Intelligence and Software Engineering. Los Altos: Morgan Kaufmann 1986

25. C. Rich: A formal representation for plans in the Programmer's Apprentice. ICJAI 1981

26. C. Rich, Y.A. Feldman: Seven Layers of Knowledge Representation and Reasoning in Support of Software Development. IEEE Trans. on Software Eng. 18(6), 451-469 (1992)

27. J.A. Robinson: A Machine-Oriented Logic Based on the Resolution Principle. J. ACM 12(1), 23-41 (1965)

28. H.B. Rubenstein, R.C. Waters: The Requirements Apprentice: Automated assistance for requirements acquisition. IEEE Trans. on Software Eng. 17, 226-240 (1991)

29. D. Setliff: Synthesizing VLSI Routing Software from Specifications. In M.R. Lowry, R.D. McCartney (eds.): Automating Software Design. Cambride, MA: AAAI/MIT Press 1991, pp. 207-226

30. D.R. Smith, M.R. Lowry: Algorithm Theories and Design Tactics. Science of Computer Programming 14:305-321 (1990)

31. D.R. Smith: KIDS -A Knowledge-Based Software Development System. In M.R. Lowry, R.D. McCartney (eds.): Automating Software Design. Cambride, MA: AAAI/MIT Press 1991, pp. 483-514

32. D.R. Smith: Constructing Specification Morphisms. Kestrel Institute Technical Report KES.U.92.1 (1992)

33. R.J. Waldinger, R.C. Lee: PROW: A step toward automatic program writing. IJCAI-69, pp. 241-252

34. R.C. Waters: Program Translation via Abstraction and Reimplementation. IEEE Trans. on Software Eng. 14(8), 1207-1228 (1988)

35. D.S Wile: Program Developments: Formal explanations of implementations. CACM 26(11): 902-911 (1983)